Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2025/26)

Lic. em Ciências da Computação

Lic. em Engenharia Informática

Grupo G05

a106936 Duarte Escairo
a106932 Luís Soares
a106856 Tiago Figueiredo

janeiro 2026

Preâmbulo

Em Cálculo de Programas pretende-se ensinar a programação de computadores como uma disciplina científica. Para isso parte-se de um repertório de *combinadores* que formam uma álgebra da programação e usam-se esses combinadores para construir programas *composicionalmente*, isto é, agregando programas já existentes.

Na sequência pedagógica dos planos de estudo dos cursos que têm esta disciplina, opta-se pela aplicação deste método à programação em Haskell (sem prejuízo da sua aplicação a outras linguagens funcionais). Assim, o presente trabalho prático coloca os alunos perante problemas concretos que deverão ser implementados em Haskell. Há ainda um outro objectivo: o de ensinar a documentar programas, a validá-los e a produzir textos técnico-científicos de qualidade.

Antes de abordarem os problemas propostos no trabalho, os grupos devem ler com atenção o anexo A onde encontrarão as instruções relativas ao *software* a instalar, etc.

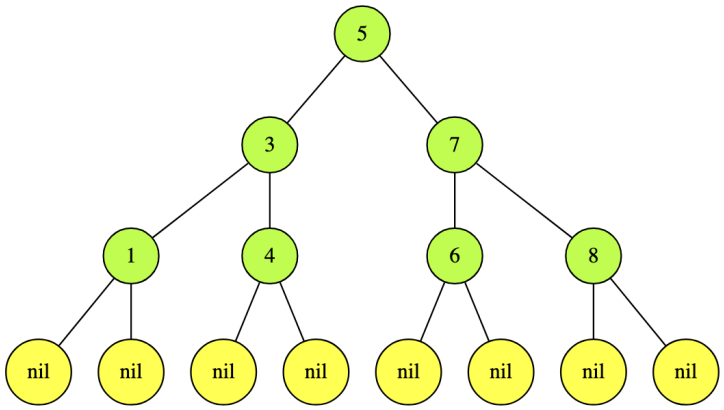
Valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes que utilizem os combinadores de ordem superior estudados na disciplina.

Avaliação. Faz parte da avaliação do trabalho a sua defesa por parte dos elementos de cada grupo. Estes devem estar preparados para responder a perguntas sobre *qualquer* dos problemas deste enunciado. A prestação *individual* de cada aluno nessa defesa oral será uma componente importante e diferenciadora da avaliação.

Problema 1

Uma serialização (ou travessia) de uma árvore é uma sua representação sob a forma de uma lista. Na biblioteca *BTree* encontram-se as funções de serialização *inordt*, *preordt* e *postordt*, que fazem as travessias *in-order*, *pre-order* e *post-order*, respectivamente. Todas essas travessias são catamorfismos que percorrem a árvore argumento em regime *depth-first*.

Pretende-se agora uma função *bforder* que faça a travessia em regime *breadth-first*, isto é, por níveis. Por exemplo, para a árvore t_1 dada em anexo e mostrada na figura a seguir,



a função deverá dar a lista

$$[5, 3, 7, 1, 4, 6, 8]$$

em que se vê como os níveis 5, depois $3, 7$ e finalmente $1, 4, 6, 8$ foram percorridos.

Pretendemos propor duas versões dessa função:

1. Uma delas envolve um catamorfismo de *BTree*s:

$$bfsLevels :: BTree\ a \to [a]$$
$$bfsLevels = concat \cdot levels$$

   Complete a definição desse catamorfismo:

$$levels :: BTree\ a \to [[a]]$$
$$levels = (\!|glevels|\!)$$

2. A segunda proposta,

$$bft :: BTree\ a \to [a]$$

   deverá basear-se num anamorfismo de listas.

Sugestão: estudar o artigo [2] cujo PDF está incluído no material deste trabalho. Quando fizer testes ao seu código pode, se desejar, usar funções disponíveis na biblioteca *Exp* para visualizar as árvores em GraphViz (formato .dot).

Justifique devidamente a sua resolução, que deverá vir acompanhada de diagramas explicativos. Como já se disse, valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes que utilizem os combinadores de ordem superior estudados na disciplina.

Problema 2

Considere a seguinte função em Haskell:

$$\begin{aligned}
&f\ x = wrapper \cdot worker\ \mathbf{where}\\
&\quad wrapper = head\\
&\quad worker\ 0 = start\ x\\
&\quad worker\ (n+1) = loop\ x\ (worker\ n)\\
&loop\ x\ [s,\quad h,\quad k,j,\quad m\ ] =\\
&\qquad [h\ /\ k+s, x\uparrow 2 * h, k * j, j+m, m+8]\\
&start\ x = [x, x\uparrow 3, 6, 20, 22]
\end{aligned}$$

Pode-se provar pela lei de recursividade mútua que $f\ x\ n$ calcula o seno hiperbólico de x, $sinh\ x$, para n aproximações da sua série de Taylor. Faça a derivação da função dada a partir da referida série de Taylor, apresentando todos os cálculos justificativos, tal como se faz para outras funções no capítulo respectivo do texto base desta UC [3].

Problema 3

Quem em Braga observar, ao fim da tarde, o tráfego onde a Avenida Clairmont Fernand se junta à N101, aproximadamente na coordenada 41°33'46.8"N 8°24'32.4"W — ver as setas da figura que se segue — reparará nas sequências imparáveis (infinitas!) de veículos provenientes dessas vias de circulação.

Mas também irá observar um comportamento interessante por parte dos condutores desses veículos: por regra, *cada carro numa via deixa passar, à sua frente, exactamente outro carro da outra via.*

Este comportamento *civilizado* chama-se *fair-merge* (ou *fair-interleaving*) de duas sequências infinitas, também designadas *streams* em ciência da computação. Seja dado o tipo dessas sequências em Haskell,

$$\textbf{data}\ Stream\ a = Cons\ (a, Stream\ a)\ \textbf{deriving}\ Show$$

para o qual se define também:

$$\text{out}\ (Cons\ (x, xs)) = (x, xs)$$

O referido comportamento civilizado pode definir-se, em Haskell, da forma seguinte:[1]

$$\begin{array}{l} fair\_merge :: (Stream\ a, Stream\ a) + (Stream\ a, Stream\ a) \to Stream\ a \\ fair\_merge = [h, k]\ \textbf{where} \\ \quad h\ (Cons\ (x, xs), y) = Cons\ (x, k\ (xs, y)) \\ \quad k\ (x, Cons\ (y, ys)) = Cons\ (y, h\ (x, ys)) \end{array}$$

Defina *fair_merge* como um **anamorfismo** de *Stream*s, usando o combinador

$$[\!(g)\!] = Cons \cdot (id \times [\!(g)\!]) \cdot g$$

e a seguinte estratégia:

- Derivar a lei **dual** da recursividade mútua,

$$[f, g] = [\!(\,[h, k]\,)\!] \quad \equiv \quad \begin{cases} out \cdot f = \mathsf{F}\ [f, g] \cdot h \\ out \cdot g = \mathsf{F}\ [f, g] \cdot k \end{cases} \tag{1}$$

  tal como se fez, nas aulas, para a que está no formulário.

- Usar (1) na resolução do problema proposto.

Justificar devidamente a resolução, que deverá vir acompanhada de diagramas explicativos.

Problema 4

Como se sabe, é possível pensarmos em catamorfismos, anamorfismos etc *probabilísticos*, quer dizer, programas recursivos que dão distribuições como resultados. Por exemplo, podemos pensar num combinador

$$pcataList :: (() + (a, b) \to \text{Dist}\ b) \to [a] \to \text{Dist}\ b$$

[1] O facto das sequências serem infinitas não nos deve preocupar, pois em Haskell isso é lidado de forma transparente por lazy evaluation.

que é muito parecido com

$$(\!|\cdot|\!) :: ((() + (a,b) \to b) \to [a] \to b$$

da biblioteca List. A principal diferença é que o gene de *pcataList* é uma função probabilística.

Como exemplo de utilização, recorde-se que $(\!|\,[zero, add]\,|\!)$ soma todos os elementos da lista argumento, por exemplo:

$$(\!|\,[zero, add]\,|\!)\ [20, 10, 5] = 35.$$

Considere-se agora a função *padd* (adição probabilística) que, com probabilidade 90% soma dois números e com probabilidade 10% os subtrai:

$$padd\ (a,b) = D\ [(a+b, 0.9), (a-b, 0.1)]$$

Se se correr

$$d4 = pcataList\ [pzero, padd]\ [20, 10, 5]\ \textbf{where}\ pzero = return \cdot zero$$

obter-se-á:

```
35  81.0%
25   9.0%
 5   9.0%
15   1.0%
```

Com base neste exemplo, resolva o seguinte

> ***Problema**: Uma unidade militar pretende enviar uma mensagem urgente a outra, mas tem o aparelho de telegrafia meio avariado. Por experiência, o telegrafista sabe que a probabilidade de uma palavra se perder (não ser transmitida) é 5%; e que, no final de cada mensagem, o aparelho envia o código* `"stop"`*, mas (por estar meio avariado), falha 10% das vezes.*
>
> *Qual a probabilidade de a palavra* `"atacar"` *da mensagem*
>
> $$words\ \texttt{"Vamos atacar hoje"}$$
>
> *se perder, isto é, o resultado da transmissão ser* `["Vamos","hoje","stop"]`*? E a de seguirem todas as palavras, mas faltar o* `"stop"` *no fim? E a da transmissão ser perfeita?*

Responda a estas perguntas encontrando *gene* tal que

$$transmitir = pcataList\ gene$$

descreve o comportamento do aparelho. Justificar devidamente a resolução, que deverá vir acompanhada de diagramas explicativos.

Anexos

A Natureza do trabalho a realizar

Este trabalho teórico-prático deve ser realizado por grupos de 3 alunos. Os detalhes da avaliação (datas para submissão do relatório e sua defesa oral) são os que forem publicados na página da disciplina na *internet*.

Recomenda-se uma abordagem participativa dos membros do grupo em **todos** os exercícios do trabalho, para assim poderem responder a qualquer questão colocada na *defesa oral* do relatório.

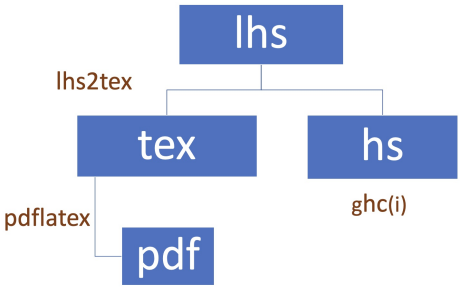
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita "literária" [1], cujo princípio base é o seguinte:

> *Um programa e a sua documentação devem coincidir.*

Por outras palavras, o **código fonte** e a **documentação** de um programa deverão estar no mesmo ficheiro.

O ficheiro `cp2526t.pdf` que está a ler é já um exemplo de programação literária: foi gerado a partir do texto fonte `cp2526t.lhs`[1] que encontrará no material pedagógico desta disciplina descompactando o ficheiro `cp2526t.zip`.

Como se mostra no esquema abaixo, de um único ficheiro (*lhs*) gera-se um PDF ou faz-se a interpretação do código Haskell que ele inclui:



Vê-se assim que, para além do GHCi, serão necessários os executáveis pdflatex e lhs2TeX. Para facilitar a instalação e evitar problemas de versões e conflitos com sistemas operativos, é recomendado o uso do Docker tal como a seguir se descreve.

B Docker

Recomenda-se o uso do container cuja imagem é gerada pelo Docker a partir do ficheiro `Dockerfile` que se encontra na diretoria que resulta de descompactar `cp2526t.zip`. Este container deverá ser usado na execução do GHCi e dos comandos relativos ao LaTeX. (Ver também a `Makefile` que é disponibilizada.)

Após instalar o Docker e descarregar o referido zip com o código fonte do trabalho, basta executar os seguintes comandos:

```
$ docker build -t cp2526t .
$ docker run -v ${PWD}:/cp2526t -it cp2526t
```

NB: O objetivo é que o container seja usado *apenas* para executar o GHCi e os comandos relativos ao LaTeX. Deste modo, é criado um *volume* (cf. a opção `-v ${PWD}:/cp2526t`) que permite que a diretoria em que se encontra na sua máquina local e a diretoria `/cp2526t` no container sejam partilhadas.

Pretende-se então que visualize/edite os ficheiros na sua máquina local e que os compile no container, executando:

[1] O sufixo 'lhs' quer dizer *literate Haskell*.

```
$ lhs2TeX cp2526t.lhs > cp2526t.tex
$ pdflatex cp2526t
```

lhs2TeX é o pre-processador que faz "pretty printing" de código Haskell em LaTeX e que faz parte já do container. Alternativamente, basta executar

```
$ make
```

para obter o mesmo efeito que acima.

Por outro lado, o mesmo ficheiro `cp2526t.lhs` é executável e contém o "kit" básico, escrito em Haskell, para realizar o trabalho. Basta executar

```
$ ghci cp2526t.lhs
```

Abra o ficheiro `cp2526t.lhs` no seu editor de texto preferido e verifique que assim é: todo o texto que se encontra dentro do ambiente

```
\begin{code}
...
\end{code}
```

é seleccionado pelo GHCi para ser executado.

C Em que consiste o TP

Em que consiste, então, o *relatório* a que se referiu acima? É a edição do texto que está a ser lido, preenchendo o anexo G com as respostas. O relatório deverá conter ainda a identificação dos membros do grupo de trabalho, no local respectivo da folha de rosto.

Para gerar o PDF integral do relatório deve-se ainda correr os comando seguintes, que actualizam a bibliografia (com BibTeX) e o índice remissivo (com `makeindex`),

```
$ bibtex cp2526t.aux
$ makeindex cp2526t.idx
```

e recompilar o texto como acima se indicou. (Como já se disse, pode fazê-lo correndo simplesmente `make` no container.)

No anexo F disponibiliza-se algum código Haskell relativo aos problemas que são colocados. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

Deve ser feito uso da programação literária para documentar bem o código que se desenvolver, em particular fazendo diagramas explicativos do que foi feito e tal como se explica no anexo D que se segue.

D Como exprimir cálculos e diagramas em LaTeX/lhs2TeX

Como primeiro exemplo, estudar o texto fonte (lhs) do que está a ler[1] onde se obtém o efeito seguinte:[2]

$$\begin{array}{ll} & id = \langle f, g\rangle \\ \equiv & \quad \{\ \text{universal property}\ \} \end{array}$$

[1] Procure e.g. por `"sec:diagramas"`.

[2] Exemplos tirados de [3].

$$
\begin{cases} \pi_1 \cdot id = f \\ \pi_2 \cdot id = g \end{cases}
$$

$$
\equiv \quad \{ \text{ identity } \}
$$

$$
\begin{cases} \pi_1 = f \\ \pi_2 = g \end{cases}
$$

$$
\square
$$

Os diagramas podem ser produzidos recorrendo à *package* xymatrix, por exemplo:

$$
\begin{array}{ccc}
\mathbb{N}_0 & \xleftarrow{\text{in}} & 1 + \mathbb{N}_0 \\
{\scriptstyle (\!|g|\!)}\downarrow & & \downarrow{\scriptstyle id + (\!|g|\!)} \\
B & \xleftarrow[g]{} & 1 + B
\end{array}
$$

E O mónade das distribuições probabilísticas

Mónades são functores com propriedades adicionais que nos permitem obter efeitos especiais em programação. Por exemplo, a biblioteca Probability oferece um mónade para abordar problemas de probabilidades. Nesta biblioteca, o conceito de distribuição estatística é captado pelo tipo

$$
\textbf{newtype}\ \text{Dist}\ a = D\ \{unD :: [(a, ProbRep)]\} \tag{2}
$$

em que $ProbRep$ é um real de 0 a 1, equivalente a uma escala de 0 a 100%.

Cada par (a, p) numa distribuição $d :: \text{Dist}\ a$ indica que a probabilidade de a é p, devendo ser garantida a propriedade de que todas as probabilidades de d somam 100%. Por exemplo, a seguinte distribuição de classificações por escalões de A a E,

A ▬ 2%
B ▬▬▬ 12%
C ▬▬▬▬▬▬▬▬ 29%
D ▬▬▬▬▬▬▬▬▬▬▬ 35%
E ▬▬▬▬▬▬▬ 22%

será representada pela distribuição

$$
\begin{array}{l}
d1 :: \text{Dist}\ Char \\
d1 = D\ [('\texttt{A}', 0.02), ('\texttt{B}', 0.12), ('\texttt{C}', 0.29), ('\texttt{D}', 0.35), ('\texttt{E}', 0.22)]
\end{array}
$$

que o GHCi mostrará assim:

```
'D'  35.0%
'C'  29.0%
'E'  22.0%
'B'  12.0%
'A'   2.0%
```

É possível definir geradores de distribuições, por exemplo distribuições *uniformes*,

$$
d2 = uniform\ (words\ \texttt{"Uma frase de cinco palavras"})
$$

isto é

```
   "Uma"  20.0%
 "cinco"  20.0%
    "de"  20.0%
 "frase"  20.0%
"palavras"  20.0%
```

distribuição *normais*, eg.

$$d3 = normal\ [10\,..\,20]$$

etc.[1] Dist forma um **mónade** cuja unidade é $return\ a = D\ [(a,1)]$ e cuja composição de Kleisli é (simplificando a notação)

$$(f \bullet g)\ a = [(y, q*p)\ |\ (x,p) \leftarrow g\ a, (y,q) \leftarrow f\ x]$$

em que $g : A \to$ Dist B e $f : B \to$ Dist C são funções **monádicas** que representam *computações probabilísticas*.

Este mónade é adequado à resolução de problemas de *probabilidades e estatística* usando programação funcional, de forma elegante e como caso particular da programação monádica.

F Código fornecido

Problema 1

Árvores exemplo:

```
t1 :: BTree Int
t1 = Node (5,(Node (3,(Node (1,(Empty,Empty)),Node (4,(Empty,Empty)))),
   Node (7,(Node (6,(Empty,Empty)),Node (8,(Empty,Empty))))))
t2 :: BTree Int
t2 =
   node 1
      (node 2 (node 4 Empty Empty) (node 5 Empty Empty))
      (node 3 (node 6 Empty Empty) (node 7 Empty Empty))
t3 :: BTree Char
t3 =
   node 'A'
      (node 'B' (node 'C' (node 'D' Empty Empty) Empty) Empty)
      (node 'E' Empty Empty)
t4 :: BTree Char
t4 =
   node 'A'
      (node 'B' (node 'C' (node 'D' Empty Empty) Empty) Empty)
      Empty
t5 :: BTree Int
t5 =
   node 1
```

[1] Para mais detalhes ver o código fonte de Probability, que é uma adaptação da biblioteca PFP ("Probabilistic Functional Programming"). Para quem quiser saber mais recomenda-se a leitura do artigo [**?**].

$(node\ 2\ (node\ 4\ Empty\ Empty)\ Empty)$
$(node\ 3\ Empty\ (node\ 5\ (node\ 6\ Empty\ Empty)\ Empty))$

$node\ a\ b\ c = Node\ (a,(b,c))$

G Soluções dos alunos

Os alunos devem colocar neste anexo as suas soluções para os exercícios propostos, de acordo com o "layout" que se fornece. Não podem ser alterados os nomes ou tipos das funções dadas, mas pode ser adicionado texto ao anexo, bem como diagramas e/ou outras funções auxiliares que sejam necessárias.

Importante: Não pode ser alterado o texto deste ficheiro fora deste anexo.

Problema 1

Catamorfismo

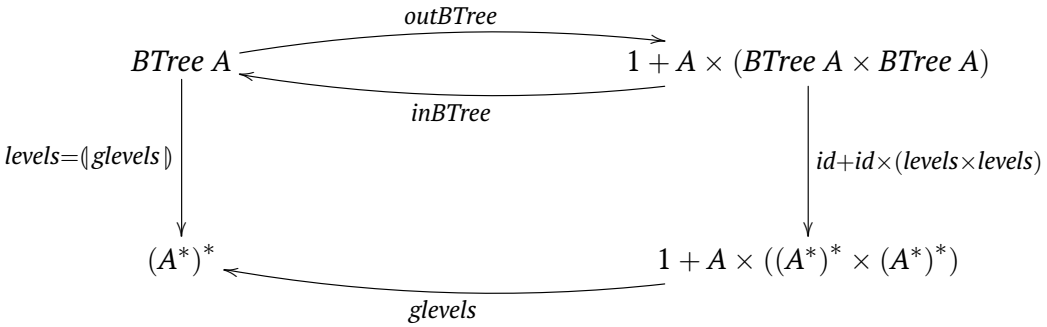
Na primeira versão proposta para a resolução do **Problema 1**, pretende-se usar um catamorfismo de *BTrees* para se fazer a travessia *in-order* em regime *breadth-first*.

Se repararmos, o resultado de aplicarmos a função *levels* a uma *BTree* é uma lista de listas, onde cada uma dessas listas internas corresponde aos valores dos nós de um nível da árvore. Ou seja, a aplicação de *levels* à árvore t_1, por exemplo, resulta na lista de listas:

$$[[5],[3,7],[1,4,6,8]]$$

Para depois obter a travessia *bforder*, basta concatenar todas as listas internas, o que é feito na função *bfsLevels* com recurso à função *concat*.

O desafio aqui está em encontrar o gene (*glevels*) do catamorfismo *levels*. Para começar, podemos representar esse catamorfismo através do seguinte diagrama:



A partir deste diagrama, percebemos que o gene do catamorfismo deverá, para obter a tal lista de listas, colocar o valor da raiz no início da lista final, seguido das listas dos níveis das subárvores esquerda e direita. Contudo essas listas dos níveis das subárvores estão também organizadas por níveis, ou seja, a primeira lista corresponde ao nível 1, a segunda ao nível 2, etc. No caso da àrvore t_1, por exemplo, a aplicação de *levels* às subárvores esquerda e direita de t_1 resulta, respectivamente, nas listas de listas:

$$[[3],[1,4]]\ e\ [[7],[6,8]]$$

Neste caso o passo final seria juntar as listas dos níveis das subárvores esquerda e direita, e colocar o valor da raiz 5 ($[5]$) no início da lista final. Para juntar as listas dos níveis das subárvores esquerda e direita, podemos definir a seguinte função auxiliar:

$$
\begin{aligned}
&\mathit{juntaListas} :: ([[a]],[[a]]) \to [[a]] \\
&\mathit{juntaListas}\ (l,[\,]) = l \\
&\mathit{juntaListas}\ ([\,],r) = r \\
&\mathit{juntaListas}\ ((a:as),(b:bs)) = (a +\!\!+ b) : \mathit{juntaListas}\ (as,bs)
\end{aligned}
$$

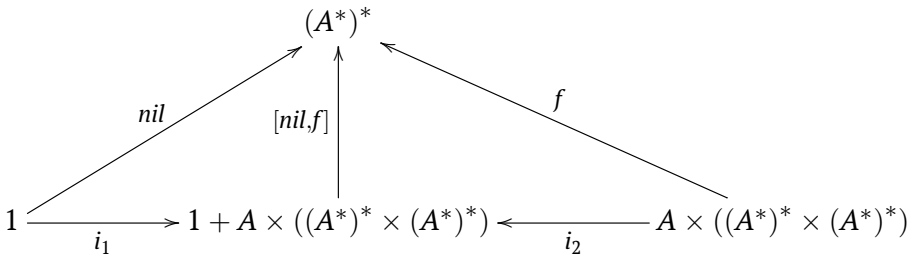
Para o exemplo da árvore t_1, a aplicação de *juntaListas* às listas de listas

$$[[3],[1,4]]\ e\ [[7],[6,8]]$$

resulta na lista de listas:

$$[[3,7],[1,4,6,8]]$$

Assim sendo, o comportamento do gene *glevels* pode ser representado por este diagrama:

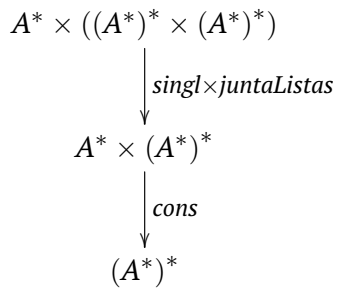


E o código que define o gene *glevels* é o seguinte:

$$
\begin{aligned}
&\mathit{glevels} :: () + (a,([[a]],[[a]])) \to [[a]] \\
&\mathit{glevels} = [\mathit{nil}, f] \\
&\quad \textbf{where}\ f\ (a,(l,r)) = [a] : \mathit{juntaListas}\ (l,r)
\end{aligned}
$$

Tal como descrito anteriormente, f coloca o valor da raiz à cabeça da lista, seguido da concatenação das listas dos níveis das subárvores esquerda e direita, com a função *juntaListas*.

Se quisermos definir *glevels* numa versão completamente *pointfree*, podemos esquematizar o lado direito da solução desta forma:



Logo a definição de *glevels* é a seguinte:

$$\mathit{glevels}' = [\mathit{nil}, \mathit{cons} \cdot (\mathit{singl} \times \mathit{juntaListas})]$$

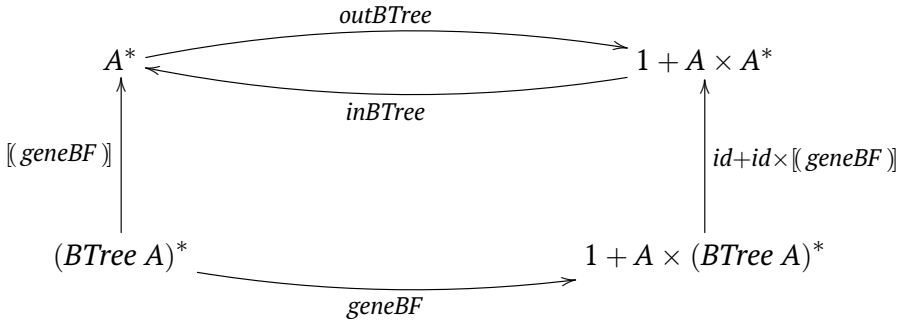
Anamorfismo

Na segunda versão proposta para a resolução do **Problema 1**, pretende-se usar um anamorfismo de listas para se fazer a travessia *in-order* em regime *breadth-first*.

Com o estudo do artigo [2], percebemos que é possível fazer a travessia *breadth-first* em *BTrees* usando uma floresta para representar os nós a visitar (neste caso uma floresta é uma lista de *BTrees*).

A ideia do algoritmo é visitar os nós da floresta, retirando o valor do nó da frente da lista, adicionando o seu valor à lista resultado (caso não seja *Empty*) e adicionando à floresta os filhos esquerdo e direito do nó visitado (caso não seja *Empty*), mas colocando-os no final da lista. Este processo repete-se até que a floresta esteja vazia.

Para representar este algoritmo como um anamorfismo, começamos por desenhar o seguinte diagrama:



Contudo, como referido anteriormente, o algoritmo usa uma floresta (lista de *BTrees*), e não um *BTree*, que é o input da função *bft*. Desta forma é necesário garantir que a *BTree* argumento seja convertida numa floresta, para poder ser usada no anamorfismo. Para isso, usamos a função *singl*, que transforma uma *BTree* numa floresta com apenas essa *BTree* (lista com apenas um elemento).

$$BTree\ A \xrightarrow{\quad singl \quad} (BTree\ A)^*$$

Assim sendo, a função *bft* é definida como a composição do anamorfismo $[\!(\ geneBF\ )\!]$ com a função *singl*, como se mostra a seguir:

$$bft = [\!(\ geneBF\ )\!] \cdot singl$$

Para o gene do anamorfismo, o comportamento é o já anteriormente descrito, ou seja, visitar os nós da floresta, retirando o valor do nó da frente da lista, e devolvendo um par com o valor do nó e a floresta a visitar. Na floresta a visitar, já estarão adicionados os filhos esquerdo e direito do nó visitado (caso não seja *Empty*), colocados no final da lista. O anamorfismo repete este processo até que a floresta esteja vazia.

$$\begin{aligned}
&geneBF :: [BTree\ a] \to () + (a, [BTree\ a]) \\
&geneBF\ [\,] = i_1\ () \\
&geneBF\ (Empty : t) = geneBF\ t \\
&geneBF\ (Node\ (a,(l,r)) : t) = i_2\ (a, t +\!\!+ [l,r])
\end{aligned}$$

Problema 2

No **Problema 2**,

Problema 3

No **Problema 3**, pretende-se definir a função *fair_merge* como um anamorfismo de *Streams*. Esta estratégia de *merging* permite que os elementos das duas *Streams* sejam intercalados de forma justa, ou seja, cada elemento de uma *Stream* é seguido por um elemento da outra *Stream*.

Tal como pedido no enunciado, vamos começar por derivar a lei dual da recursividade mútua.

$$
\begin{aligned}
& [f, g] = [\!([h, k])\!] \\
\equiv & \quad \{ \text{ Universal-ana } \} \\
& \mathsf{out} \cdot [f, g] = \mathsf{F}\ [f, g] \cdot [h, k] \\
\equiv & \quad \{ \text{ Fusão- + (2x) } \} \\
& [\mathsf{out} \cdot f, \mathsf{out} \cdot g] = [\mathsf{F}\ [f, g] \cdot h, \mathsf{F}\ [f, g] \cdot k] \\
\equiv & \quad \{ \text{ Eq- + } \} \\
& \begin{cases} \mathsf{out} \cdot f = \mathsf{F}\ [f, g] \cdot h \\ \mathsf{out} \cdot g = \mathsf{F}\ [f, g] \cdot k \end{cases} \\
\square &
\end{aligned}
$$

Fica então demonstrada a lei dual da recursividade mútua.

A partir da definição do tipo *Stream* e da função out, é possível deduzir que o functor F associado a *Stream* é o seguinte,

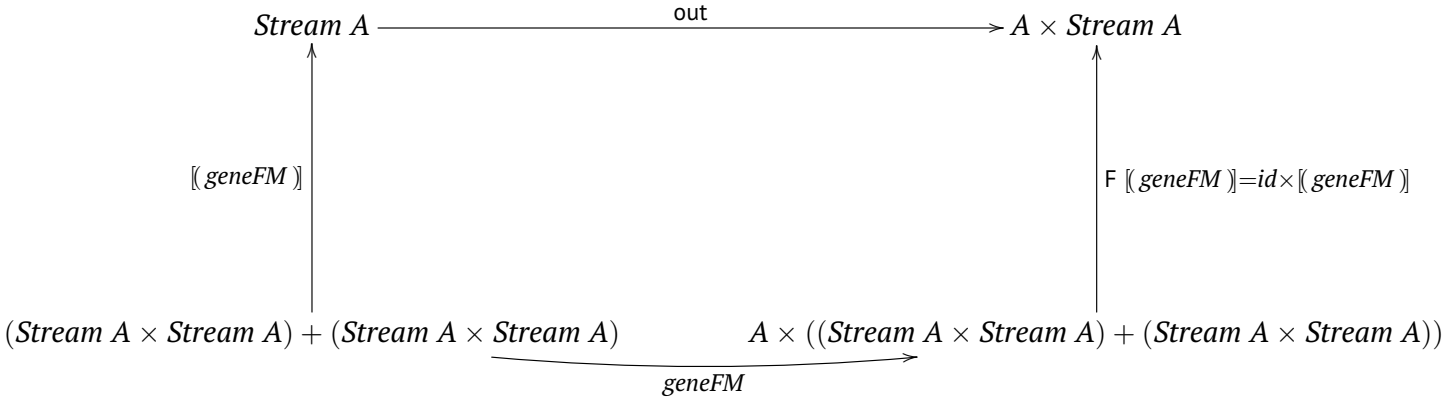
$$
\begin{aligned}
& \mathsf{F}\ X = A \times X \\
& \mathsf{F}\ g = id \times g
\end{aligned}
$$

já que uma *Stream A* é decomposta num par com o primeiro elemento de um tipo qualquer A e o resto da *Stream A*.

Tendo isto em conta, e sabendo que a função $fair\_merge'$ que queremos definir como um anamorfismo de *Streams* é do tipo

$$fair\_merge' :: (Stream\ a, Stream\ a) + (Stream\ a, Stream\ a) \to Stream\ a$$

podemos representar o anamorfismo $fair\_merge'$ através do seguinte diagrama:



Com a análise deste diagrama, e do comportamento da função *fair_merge* definida de forma mutuamente recursiva na formulação do problema neste enunciado

$$
\begin{aligned}
&fair\_merge :: (Stream\ a, Stream\ a) + (Stream\ a, Stream\ a) \to Stream\ a \\
&fair\_merge = [h, k]\ \textbf{where} \\
&\quad h\ (Cons\ (x, xs), y) = Cons\ (x, k\ (xs, y)) \\
&\quad k\ (x, Cons\ (y, ys)) = Cons\ (y, h\ (x, ys))
\end{aligned}
$$

conseguimos definir $fair\_merge'$ como um anamorfismo de *Streams*

$$fair\_merge' = [\!(\ geneFM\ )\!]$$

onde o gene $geneFM$ é definido como:

$$
\begin{aligned}
&geneFM :: (Stream\ a, Stream\ a) + (Stream\ a, Stream\ a) \to (a, (Stream\ a, Stream\ a) + (Stream\ a, Stream\ a)) \\
&geneFM\ (i_1\ (Cons\ (x, xs), y)) = (x, i_2\ (xs, y)) \\
&geneFM\ (i_2\ (x, Cons\ (y, ys))) = (y, i_1\ (x, ys))
\end{aligned}
$$

Para comprovar este resultado, podemos utilizar a lei dual da recursividade mútua que foi demonstrada anteriormente. Seja $fair\_merge$ dada por:

$$fair\_merge = [h, k]$$

E querendo demonstrar que $fair\_merge = [\!(\ geneFM\ )\!]$, onde $geneFM$ pode ser decomposto em:

$$geneFM = [g1, g2]$$

Tem-se:

$$
\begin{aligned}
& [h, k] = [\!(\ geneFM\ )\!] \\
\equiv & \quad \{\ \text{Lei dual da recursividade mútua}\ \} \\
& \begin{cases} \text{out} \cdot h = (id \times [h, k]) \cdot g1 \\ \text{out} \cdot k = (id \times [h, k]) \cdot g2 \end{cases} \\
\square &
\end{aligned}
$$

Problema 4

Introdução

De uma forma simples, o objetivo deste problema é desenhar uma função *transmitir* que descreve o comportamento de um aparelho de telegrafia avariado. Esse aparelho tenta transmitir uma mensagem palavra a palavra, mas pode falhar de forma aleatória: cada palavra pode perder-se durante a transmissão e, no final da mensagem, o envio do código "stop" também pode falhar.

Pretende-se que a função *transmitir* modele corretamente estas falhas, produzindo todas as mensagens possíveis, cada uma associada à respetiva probabilidade. Para tal, recorre-se a um catamorfismo probabilístico sobre listas, *pcataList*, cujo resultado é uma distribuição de probabilidades, representada pelo mónade Dist.

O comportamento local do aparelho é descrito por um *gene*, que define as decisões probabilísticas a tomar em cada passo da transmissão. A função $transmitir = pcataList\ gene$ separa assim o mecanismo genérico de percorrer a lista, do comportamento específico do aparelho.

Antes de apresentar a solução completa, é útil analisar cada componente que será usado.

Uso de Dist

No nosso projeto, é necessário modelar a transmissão de mensagens em que cada palavra pode falhar de forma aleatória. Para isso, precisamos de uma forma de representar todas as mensagens possíveis e associar a cada uma a sua probabilidade de ocorrência.

O mónade Dist faz exatamente isso:

$$\textbf{newtype}\ \text{Dist}\ a = D\ \{unD :: [(a, ProbRep)]\}$$

- Cada a é um possível valor (no caso, uma mensagem ou palavra transmitida).
- Cada ProbRep é a probabilidade desse valor ocorrer (um número real entre 0 e 1, sendo a soma total das probabilidades 1).

Dist forma um mónade porque:

- Tem uma operação return: $return\ a = D\ [(a, 1)]$, que gera uma distribuição determinística com probabilidade 1.
- Tem uma composição de Kleisli (bind »=): $(f \bullet g)\ a = [(y, q * p) \mid (x, p) \leftarrow g\ a, (y, q) \leftarrow f\ x]$, onde $g :: A \to \texttt{Dist B}$ e $f :: B \to \texttt{Dist C}$ são funções que representam computações probabilísticas.

Esta definição permite combinar automaticamente as probabilidades de múltiplas etapas de cálculo. Por exemplo, ao percorrer uma lista de palavras, o mónade Dist calcula todas as combinações possíveis de palavras transmitidas e perdidas, multiplicando as probabilidades associadas a cada passo, sem necessidade de ciclos explícitos ou cálculos manuais.

Desta forma, alterações no comportamento do *gene* propagam-se corretamente por toda a lista, sendo suficiente ajustar apenas as probabilidades associadas a cada decisão local.

Uso do gene

O *gene* descreve o comportamento local do aparelho de telegrafia. Para cada palavra, define quais são os resultados que podem ocorrer e com que probabilidades. Neste caso, deverá calcular o caso da perda de uma palavra ou o caso da perda da palavra final "stop".

O tipo do *gene* é:

$$gene :: () + (String, [String]) \to \text{Dist}\ [String]$$

O uso do $\cdot + \cdot$ permite separar de forma natural os dois casos distintos que surgem ao percorrer uma lista:

- **Lista vazia:** representada por $i_1\ ()$. Este caso corresponde ao fim da lista, permitindo decidir probabilisticamente se a transmissão termina corretamente ou se ocorre a falha de envio do "stop".
- **Lista não vazia:** representada por $i_2\ (String, [String])$, onde x é a cabeça da lista (palavra atual) e y é o resultado já processado da cauda. Este caso permite combinar a palavra atual com todos os resultados possíveis da cauda, aplicando as regras probabilísticas definidas.

Desta forma, o *gene* consegue tratar de forma modular e uniforme tanto o caso base da lista, quanto a transmissão de cada palavra individual, sem que a função de percorrer a lista (*pcataList*) precise de conhecer detalhes do comportamento probabilístico como: as probabilidades definidas ou possíveis decisões probabilísticas complexas (por exemplo, múltiplas opções para cada palavra ou eventos condicionais).

Uso de pcataList

O catamorfismo *pcataList* percorre a lista de palavras de forma recursiva, aplicando a função *gene* a cada passo. Esta abordagem permite separar a lógica de percorrer a lista da lógica de transmissão probabilística, tornando o comportamento do aparelho mais fácil de modelar. Sabendo a declaração de *pcataList* (dada no enunciado):

$$pcataList :: (() + (a, b) \to \text{Dist } b) \to [a] \to \text{Dist } b$$

E que em listas os catamorfismos devem ter em conta os casos de lista vazia e não vazia, é possível deduzir a definição de *pcataList* como sendo:

$$\begin{aligned} &pcataList\ gene\ [\,] = gene\ (i_1\ ()) \\ &pcataList\ gene\ (x : xs) = \mathbf{do} \\ &\quad y \leftarrow pcataList\ gene\ xs \\ &\quad gene\ (i_2\ (x, y)) \end{aligned}$$

Nesta definição:

- No caso de lista vazia, *gene* recebe $i_1\ ()$, permitindo decidir probabilisticamente se o processo termina.
- No caso de lista não vazia, a cauda da lista é processada primeiro recursivamente, produzindo y, que é então combinado com a cabeça x através de $gene\ (i_2\ (x, y))$.

Para ilustrar o seu funcionamento, considere a mensagem:

["hi","hi again","bye"]

A aplicação de *pcataList gene* a esta lista é definida recursivamente, seguindo a estrutura do catamorfismo sobre listas. Os passos da execução, são:

- **Passo 1: lista completa ["hi","hi again","bye"]**

$$\begin{aligned} &pcataList\ gene\ [\texttt{"hi"}, \texttt{"hi again"}, \texttt{"bye"}] = \mathbf{do}\ \{ \\ &\quad y \leftarrow pcataList\ gene\ [\texttt{"hi again"}, \texttt{"bye"}]; \\ &\quad gene\ (i_2\ (\texttt{"hi"}, y))\} \end{aligned}$$

  O valor y representa uma distribuição de todos os resultados possíveis da cauda ["hi again","bye"].

- **Passo 2: cauda ["hi again","bye"]**

$$\begin{aligned} &pcataList\ gene\ [\texttt{"hi again"}, \texttt{"bye"}] = \mathbf{do} \\ &\quad y1 \leftarrow pcataList\ gene\ [\texttt{"bye"}]; \\ &\quad gene\ (i_2\ (\texttt{"hi again"}, y1)) \end{aligned}$$

  Cada resultado y_1 da chamada à cauda anterior é combinado com "hi again" através do *gene*, propagando todas as combinações possíveis com as probabilidades corretas.

- **Passo 3: cauda ["bye"]**

$$\begin{aligned} &pcataList\ gene\ [\texttt{"bye"}] = \mathbf{do} \\ &\quad y2 \leftarrow pcataList\ gene\ [\,]; \\ &\quad gene\ (i_2\ (\texttt{"bye"}, y2)) \end{aligned}$$

Cada resultado y_2 do caso base é combinado com "bye" através do *gene*, multiplicando as probabilidades de cada etapa.

- **Passo 4: caso base []**

$$pcataList\ gene\ [\,] = gene\ (i_1\ ())$$

Produz a distribuição inicial que representa as possíveis terminações da mensagem, de acordo com o comportamento probabilístico do *gene*.

A partir deste caso base, o catamorfismo combina sucessivamente cada palavra com todos os resultados possíveis da sua cauda, multiplicando as probabilidades locais. Deste modo, o valor

$$pcataList\ gene\ [\texttt{"hi"}, \texttt{"hi again"}, \texttt{"bye"}]$$

corresponde a uma única distribuição final que contém todas as mensagens possíveis resultantes da transmissão, já com as probabilidades corretamente combinadas.

Index

References

[1] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[2] Chris Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 131–136. ACM, 2000.

[3] J.N. Oliveira. Program Design by Calculation, 2024. Draft of textbook in preparation. First version: 1998. Current version: Sep. 2024. Informatics Department, University of Minho (pdf).